PARALLEL EXECUTION OF CSOUND

John ffitch

Department of Computer Science
University of Bath
Bath, UK
jpff@cs.bath.ac.uk

ABSTRACT

A design for a shared-memory multicore evaluation system for Csound is developed, that maintains the same semantics as the sequential system, but is capable of scaling to a number of cores. This system is based on earlier parallel processor technology and also on methods of instruction allocation in RISC computers. The user need not be aware of the scheme or take any special actions, and it is totally transparent to the collection of legacy pieces.

1. INTRODUCTION

It is becoming impossible to buy a computer with only one processor or core, but the software in general use is often older than this architectural change, and can only make use of one processor at a time. Rather than computers getting faster, and taking less time to the end of a task the user is offered the ability to do two tasks at the same time, both at the previous slow speed.

Like all specialist software areas, audio processing must face the implications of at least multi-core computing systems. Elsewhere the use of array accelerator processors and High Performance Computing has been considered[2, 4] and there has been a panel on the general issue[13]. In this paper we consider one such 'legacy' system from the field of computer music, the Csound system[1]. The original system was written over twenty years ago, and largely rewritten fifteen years later[3], but substantially before the advent of the ubiquitous multi-core desktop or portable computer. The redesign work took no input from parallelism.

The use of a published API has allowed for the creation of more complex user interfaces, and links to other audio processing systems, but as mentioned in the opening paragraph, these do not have a substantial effect on time-to-completion. We are concern here with internal processes, below the level of an API or indeed user actions, aimed as a shared-memory multi-processor computer.

2. CSOUND5

Csound is a member of the MusicV family of audio processing systems, and has the separation between an *orchestra* and a *score*. The orchestra is a collection of *instruments*, each then being a sequence of unit generators. The score is largely an event list, that dictates which instrument is to be played when, and can pass parameters such as amplitude or pitch to the instance. The unit generators have two components, and initialiser and a performer. On starting to play an instrument all the unit generator initialisers are called, and then at a control rate (called the k-rate) the performers are called repeatedly until the end of the note. A few unit generators also have destructors, but this is rare.

What is important for this paper is the semantics; in any control cycle the instruments are performed in increasing numerical order[1], and this ensures for example that signals created can be passed to an effects instrument in the same cycle.

Instruments can communicate with each other in a variety of ways. The simplest is via a global variable (either a scalar or an audio value), but there is also the ZAK bus system, and software busses to pass data via a controlling program. This last mechanism is a consequence of the transformation of Csound to a library that was a major component of the Csound5 rewrite.

Any attempt at parallel rendering of audio in Csound must maintain this semantics, and user pieces rely on it, and an immutable principle of Csound is that we must not break existing pieces.

3. PREVIOUS PARALLELISM PROJECTS

Naturally this is not the first attempt at a parallel execution system for an existing program. We describe here previous Csound approaches, and some experiments from long ago on which we are building our model.

[1] Instruments are numbered, or if named numbers are picked to present them

3.1. Csound and Parallelism

The Extended Csound system[12] started as an implementation of Csound for the SHARC that was capable of real-time synthesis, but later when there was insufficient processing power multiple processors were use. The method of parallel execution was to distribute the instruments to processors, and outlaw global variables. This was relaxed in two cases; passing audio to an effects instruments and passing the output of that instrument to a reverberation instrument. By adding a cycle of latency for each of these there is no conflicts in causality. This is achieved at the price of restricting the orchestra. While this was acceptable for the target applications it does not scale to our desire for preserving the semantics. It did however demonstrate that parallelism could be used, and is one of the spurs for the current work.

Mention should also be made of some experimental code in Csound5, mainly due to Stephen Yi, to create multiple threads, and distribute the work in the k-rate dispatcher. This also does not take account of the semantics but is a mechanism we can use with additional care. The process of the necessary care is described in section 3.2.

3.2. The Bath LISP Machine and Parallel Algebra

Marti and Fitch in the 1980s worked on a parallel LISP machine[7], which used the functional-like structure of the LISP language for parallelism. At the heart of this work was the analysis of programs for side-effects that could stop the parallel evaluation. Marti[10] developed techniques for labelling each function with a quad structure, listing global values read, global values written, and global values both read and written. The last field was a "too hard" flag to allow for operations like RPLACA which could change anything and could not be tracked. The scheme was amplified by Fitch[5] leading to the classification of all operations in a substantial algebra system[6].

It is a matter of history that about 1990 the impetus behind parallel computing slowed and LISP fell out of fashion, and this work was halted. But this is what we need for Csound; analysis of instruments to determine when they are independent and when they require sequentialisation. With that information we can tackle the semantic issues.

4. GRANULARITY

The general plan is to use the POSIX pthreads mechanism to evaluate as many sections as we have threads in parallel, controlled by a dispatcher that has access to the dependences between the sections (section 5). In order to do this we must first consider the issue of granularity; how large should the sections be?

There are a number of obvious units for the granularity, such as the instrument, unit generator or internal function. The last of these is likely to involve a great amount of internal rewriting, as each unit generator would need modification to be aware of the parallel system, and for this reason is not considered further here.

The use of the unit generator as the grain has many attractions, and is likely to reveal many opportunities for parallelism. However the number of instructions obeyed in a single control cycle by an individual unit generator is often small – in many cases less than 10o instructions, and for these the cost of switching thread would dwarf any savings. It would be possible to collect sets of unit generators into superunits, but at present this seems a complication too far.

Our preferred granularity item is the instrument. This fits well with the instrument dispatch code, and requires only small areas of code modification.

Whatever granularity we choose it is necessary to have a measure of the computational cost of code. Even at the higher level some instruments are only two unit generators; for example an oscillator and envelope, and we need to avoid the overheads of threads in such cases.

To this end we have been building a database of unit generator costs. For many of them there are three kinds of costs – initialisations, costs for each control cycle and costs for each sample. By running Csound orchestras with two different *kr* values under `valgrind`, and some simple algebra, it is possible to calculate these costs in instructions counts. The data collection is being done on one architecture, Intel i586, but we are assuming that these figures will transfer sufficiently well. At present the process of collecting these numbers is largely manual, but it should be possible to reduce this to scripts, and then if necessary the calculations could be done for a variety of machines. We are taking not account of caching, memory stalls or other advanced architectural features.

Opcode	init	Audio	Control
table.a	93	23.063	43.998
table.k	93	0	45
butterlp	9	29.005 4	5.478
butterhi	19	30.000	35
butterbp	20	30	71

Table 1. Costs of a few opcodes.

Of course no all unit generators have this simple cost structure, but in our initial experiments we are taking 'average' behaviour (see table 1). Again if this proves insufficient it could be extended; it is clear already that this will need extensions for hopping phase vocoding. It is important to note that these calculations are offline, and used to guide the allocation of instruments to threads. Also as Csound as a language is low in looping structures many of the issues of the static estimation of run time do not arise[8].

5. SYNCHRONISATION

One of the features of Csound5 has been experiments in providing a new Bison-based parser, the work of ffitch and Yi. We are extending this parser to annotate each instrument with the external variables it uses. As there is little in the way of function calls, and certainly none to runtime values, this is considerably easier that in more general language systems. In the same fashion as the LISP compilation schemes referred to above we record the reading of global values, writing to global variables, reading and writing f-tables and a flag for instruments whose behaviour cannot be fully determined, for example by using computed ZAK locations. With this can construct a partial ordering of instruments. Central to this the semantics of increasing instrument numbers, but this constraint is only necessary when there is some activity with externals. What this annotation gives is when it is semanticly safe to run two instrument instances at the same in any cycle.

```
instr 1
  a1 oscil p4, p5, 1
     out   a1
endin
instr 2
  gk oscil p4, p5, 1
endin
instr 3
 a1  oscil gk, p5, 1
      out  a1
endin
```

Figure 1. A simple Orchestra.

For example, in the simple orchestra of figure 1 instrument 1 is independent of both instruments 2 and 3 (apart from the `out` opcode of which more later in section 6). On the other hand instrument 2 must run to completion before instrument 3, as it gives a value to a global read by instrument 3. Any number of instrument 3 instances can run at the same time but instances of instrument 2 need some care, as we must maintain the same order as a single threaded system. Our analysis will yield the results of figure 2 to encapsulate this.

From the analysis of figure 2 we can generate the partial ordering graph shown in figure 3. Any chain must obey the increasing instrument order. It is this partial ordering that is used in the dispatcher.

6. OTHER RENDEZVOUS POINTS

There are other kinds of rendezvous point, one of which was mentioned above in the `out` family of opcodes. These are accumulators of audio, where the results of the various instrument instances are added into the output bus. There is

```
Instr1: [r:{};   w:{};   easy]
Instr2: [r:{};   w:{gk}; easy]
Instr3: [r:{gk}; w:{};   easy]
```

Figure 2. Analysis of simple orchestra.

no required order for this, but there is a need for an exclusion zone so the adding is not interrupted. This kind of synchronisation is best handled by protecting the code with a spin-lock, as the computation time is small and the chance of clashes not very high. Similarly the allocation of memory needs protection, and any code in Csound that is allocating resources during the performance cycle.

In practice there a few other of these, and finding them all is rather similar to the previous code-review to seek out non-reentrant section. A clear example is the set of bus and channel operations, some of which are always correct (those that read only) and some that need protection as they update global structures. It is an unfortunate fact that finding all these occurrences is a protracted and error-prone activity.

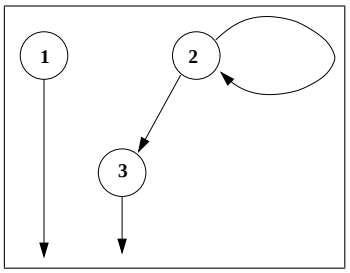

Figure 3. Partial Ordering from Analysis.

7. IMPLEMENTATION

The heart of the implementation is in the instrument dispatcher. The original code maintains a list of instances that must be run in the performance cycle. In pseudo code this is

```
until finished
  merge new instances into chain
  foreach instance in chain
    run one cycle of instance
  remove instances that have completed
end
```

The new dispatcher is conceptually the same with distributing the instances across the N processors, with the decision to run an instance on a different thread is governed by the partial order, and so becomes “run the next instance which is available if any”. The details of the dispatcher are complex and not easy to describe, but the whole process is directly analogous to the instruction scheduling algorithm of Muchnick and Gibbons[11], with ideas from VLIW processors. It is a bin-packing problem with constraints.

The set of instances (notes) is not maintained as a single chain, but as a directed acyclic graph (DAG) reflecting the partial ordering from the analysis. Each performance cycle becomes a loop of minor cycles, when an attempt is made to fill N thread-slots available. We know that we can allocate any instance in the DAG that has no prerequisites; these are the heads of the DAG. In making this allocation we can use the granularity measures to determine if an instrument-instance justifies the overhead of thread-swap, and so may run a small number of lightweight instances sequentially on a single thread, to balance the load. Some of the work having been dispatched, the DAG state is annotated to indicate the work already dispatched, and then we wait for all the threads to complete[2]. This indicates the need for another minor cycle. In this way the semantics is preserved.

The problems of deleting and adding instances has a new difficulty. In the case of simple non-dependent instruments this is trivial, but the problem of dynamic graphs is a complex one (see for example [9]), and despite the allure of $O(\log(\log(n))$ time we expect that complete reconstruction from a list will be sufficient. It is worthy of note that the number of instances in not very great.

8. CONCLUSIONS

The main concern of this paper is the design of an incremental scheme to make use of multicore computers that have shared memory. We have based this design strongly on previous computer science concepts from parallel LISP through VLIW and instruction scheduling; and importantly this system does not change the user's behaviour, except that it should be possible to complete more synthesis in real time. We believe that this schema should be sufficiently efficient to be the normal dispatcher for Csound.

There are potential problems that can be forseen; it is common to add audio signals into a global variable for reverberating, and this will show as sole use of the variable, and so force sequentiality. In fact as adding is commutative, all that is needed is a spin-lock on the actual adding. We expect that this can be handled automatically by the analyser adding spin-locks.

The implementation is not sufficiently robust to make detailed measurements on performance, but initial experiments show speedups on a dual processor of between 0.9 and 0.6 times.

The basic methodology should be applicable to any on the MusicV family of languages, as should the underlying premise to other systems – the premise that compiler analysis is at the heart of parallel execution, and we cannot expect users to change their behaviour just because the engineering requirement indicate multicore.

[2]Rather than wait at the end of every minor cycle we could start new valid instances immediately; experimentation will see if this is necessary

9. REFERENCES

[1] R. J. Boulanger, Ed., *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February 2000.

[2] R. W. Dobson, J. P. ffitch, and R. J. Bradford, "High Performance Audio Computing – A Position Paper," in *Proceedings of the 2008 ICMC*. SARC, Belfast: ICMA and SARC, 2008, pp. 213–216.

[3] J. P. ffitch, "The Design of Csound5," in *LAC2005*. Karlsruhe, Germany: Zentrum für Kunst und Medientechnologie, April 2005, pp. 37–41.

[4] J. P. ffitch, R. W. Dobson, and R. J. Bradford, "The imperative for high-performance audio computing," 2009.

[5] J. P. Fitch, "A loosely coupled parallel LISP execution system," in *The Design and Application of Parallel Digital Processors*, ser. IEE Conference Publication, vol. 298. IEE, 1988, pp. 128–133.

[6] ——, "Can REDUCE be run in parallel?" in *Proceedings of ISSAC89, Portland, Oregon*, SIGSAM. ACM, July 1989, pp. 155–162.

[7] J. P. Fitch and J. B. Marti, "The Bath Concurrent LISP machine," in *Proceedings of EUROCAL 1983*, ser. Lecture Notes in Computer Science, vol. 162, 1984, pp. 78–90.

[8] ——, "The static estimation of runtime," University of Bath Computing Group, Tech. Rep. 89–18, 1989.

[9] J. Holm, K. de Lichtenberg, and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *J. ACM*, vol. 48, no. 4, pp. 723–760, 2001.

[10] J. B. Marti, "Compilation techniques for a control-flow concurrent lisp system," in *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*. New York, NY, USA: ACM, 1980, pp. 203–207.

[11] S. S. Muchnick and P. B. Gibbons, "Efficient instruction scheduling for a pipelined architecture," *SIGPLAN Not.*, vol. 39, no. 4, pp. 167–174, 2004.

[12] B. Vercoe, "Extended Csound," in *On the Edge*, ICMA. ICMA and HKUST, 1996, pp. 141–142.

[13] D. Wessel, "Reinventing Audio and Music Computation for Many-Core Processors," in *Proceedings of the 2008 ICMC*. SARC, Belfast: ICMA and SARC, 2008.